Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

2

RT DOCUMENTATION PAGE

1a. Un

1b. RESTRICTIVE MARKINGS

2a.

3. DISTRIBUTION/AVAILABILITY OF REPORT
Unlimited

2b.

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
TR 91-1250

5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION
Cornell University

6b. OFFICE SYMBOL *(If applicable)*

7a. NAME OF MONITORING ORGANIZATION
Office of Naval Research

6c. ADDRESS *(City, State, and ZIP Code)*
Department of Computer Science
Upson Hall, Cornell University
Ithaca, NY 14853

7b. ADDRESS *(City, State, and ZIP Code)*
800 North Quincy Street
Arlington, VA 22217-5000

8a. NAME OF FUNDING/SPONSORING ORGANIZATION
Office of Naval Research

8b. OFFICE SYMBOL *(If applicable)*

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
N00014-91-J-1219

8c. ADDRESS *(City, State, and ZIP Code)*
800 North Quincy Street
Arlington, VA 22217-5000

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO

11. TITLE *(Include Security Classification)*
Paralex: An Environment for Parallel Programming in Distributed Systems

12. PERSONAL AUTHOR(S)
Ozalp Babaoglu, Lorenzo Alvisi, Alessandro Amoroso, Renzo Davoli, Luigi Alberto Giachini

13a. TYPE OF REPORT
Interim

13b. TIME COVERED
FROM ________ TO ________

14. DATE OF REPORT *(Year, Month, Day)*
1991 December 7

15. PAGE COUNT
19

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Modern distributed systems consisting of powerful workstations and high-speed interconnection networks are an economical alternative to special-purpose super computers. The technical issues that need to be addressed in exploiting the parallelism inherent in a distributed system include heterogeneity, high-latency communication, fault tolerance and dynamic load balancing. Current software systems for parallel programming provide little or no automatic support towards these issues and require users to be experts in fault-tolerant distributed computing. The Paralex system is aimed at exploring the extent to which the parallel application programmer can be liberated from the complexities of distributed systems. Paralex is a complete programming environment and makes extensive use of graphics to define, edit, execute and debug parallel scientific applications. All of the necessary code for distributing the computation across a network and replicating it to achieve fault tolerance and dynamic load balancing is automatically generated by the system. In this paper we give an overview of Paralex and present our experiences with a prototype implementation.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
☒ UNCLASSIFIED/UNLIMITED □ SAME AS RPT □ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION

22a. NAME OF RESPONSIBLE INDIVIDUAL
Fred B. Schneider

22b. TELEPHONE *(Include Area Code)*
(607) 255-9221

22c. OFFICE SYMBOL

83 APR edition may be used until exhausted.
All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

[illegible]
for public release and [illegible]
distribution is unlimited.

Paralex: An Environment for Parallel Programming in Distributed Systems[*]

Özalp Babaoğlu *Lorenzo Alvisi*

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501 (USA)

Alessandro Amoroso *Renzo Davoli* *Luigi Alberto Giachini*

Department of Mathematics
University of Bologna
40127 Bologna (Italy)

December 7, 1991

Abstract

Modern distributed systems consisting of powerful workstations and high-speed interconnection networks are an economical alternative to special-purpose super computers. The technical issues that need to be addressed in exploiting the parallelism inherent in a distributed system include heterogeneity, high-latency communication, fault tolerance and dynamic load balancing. Current software systems for parallel programming provide little or no automatic support towards these issues and require users to be experts in fault-tolerant distributed computing. The Paralex system is aimed at exploring the extent to which the parallel application programmer can be liberated from the complexities of distributed systems. Paralex is a complete programming environment and makes extensive use of graphics to define, edit, execute and debug parallel scientific applications. All of the necessary code for distributing the computation across a network and replicating it to achieve fault tolerance and dynamic load balancing is automatically generated by the system. In this paper we give an overview of Paralex and present our experiences with a prototype implementation.

[*] This work was supported in part by the Commission of European Communities under ESPRIT Programme Basic Research Action Number 3092 (Predictably Dependable Computing Systems), the United States Office of Naval Research under contract N00014-91-J-1219, IBM Corporation and the Italian Ministry of University, Research and Technology.

1 Introduction

There is general agreement that significant future increases in computing power will be possible only through exploiting parallelism. One of the many advantages of distributed systems is their potential for true parallel execution across the multiple processing elements. In fact, the amount of raw computing power that is present in a typical modern distributed system with dozens, if not hundreds, of general-purpose workstations may be comparable to an expensive, special-purpose super computer. Thus, it is tempting to try to harness the massive parallelism available in these systems for single, compute-intensive applications. There are, however, several obstacles that remain before networks of workstations can become "a poor man's supercomputer."

Distributed systems differ from special-purpose parallel computers in that they

- exhibit total asynchrony with respect to computation and communication,
- communicate over relatively low-bandwidth, high-latency networks,
- lack architectural and linguistic homogeneity,
- exhibit increased probability of communication and processor failures, and
- fall under multiple administrative domains.

As a consequence, developing parallel programs in such systems requires expertise not only in distributed computing, but also in fault tolerance. A large number of interesting applications (e.g., genome analysis) require days or weeks of computations on a network with dozens of workstations [24]. In these applications, many hours of computation can be wasted not only if there are genuine hardware failures, but also if one of the processors is turned off, rebooted or disconnected from the network. Given that the most common components of a distributed system are workstations and that they are typically under the control of several independent administrative domains (typically individuals who "own" them), these events are much more plausible and frequent than real hardware failures.

We claim that current software technology for parallel programming in distributed systems is comparable to assembly language programming for traditional sequential systems — the user must resort to low-level primitives to accomplish data encoding/decoding, communication, remote execution, synchronization, failure detection and recovery. It is our belief that reasonable technologies already exist to address each of these problems individually. What remains a challenge is the task of integrating these technologies in software support environments that permit easy development of reliable applications to exploit the parallelism and fault tolerance offered by distributed systems.

The Paralex project has been undertaken to explore the extent to which the parallel application programmer can be isolated from the complexities of distributed systems. Our goal is to realize an environment that will encompass all phases of the programming activity and provide automatic support for distribution, fault tolerance and heterogeneity in distributed and parallel applications. Paralex makes extensive use of graphics for expressing computations, controlling execution and debugging. The programming paradigm supported by Paralex promotes the view of parallel computations as collages of ordinary sequential programs. The "glue" necessary for combining computations consists of interdependencies and data flow relations that are expressed in a natural way using a graphical notation. In the limit, interesting new parallel programs can be constructed by reusing existing sequential software and without having to rewrite a single line of traditional code. As such, Paralex also addresses the issue of "software reusability" [16].

The rest of the paper is organized as follows. The next section defines the programming model supported by Paralex. In Section 3 we give an overview of the principal components of Paralex and illustrate the user interface through examples. Section 4 is a description of how Paralex uses passive replication to automatically render programs fault tolerant. A novel use of replication to also implement dynamic load balancing is discussed in Section 5. Preliminary performance results obtained from the current prototype are reported in Section 6. Paralex is put in perspective with respect to related work in Section 7. Section 8 discusses some of our design decisions and directions for future work. Section 9 concludes the paper.

2 The Paralex Programming Paradigm

The choices made for programming paradigm and notation are fundamental in harnessing parallelism in a particular application domain [17]. The programming paradigm supported by Paralex can be classified as static data flow [1]. A Paralex program is composed of *nodes* and *links*. Nodes correspond to computations (functions, procedure, programs) and the links indicate flow of (typed) data. Thus, Paralex programs can be thought of as directed graphs (and indeed are visualized as such on the screen) representing the data flow relations plus a collection of ordinary sequential code fragments to indicate the computations.

Unlike classical data flow, nodes of a Paralex program carry out significant computations. This so-called *large-grain* data flow model [6] is motivated by the high-latency, low-bandwidth network that is available for communication in distributed systems. Only by keeping the communication-to-computation ratio to reasonable levels can we expect reasonable performance from parallel applications in such systems. It is up to the user to adopt an appropriate definition for "large grain" in decomposing an application to its parallel components for a particular system.

While extremely simple, the above programming paradigm has several desirable properties. First, application parallelism is explicit in its notation — all nodes that have no data dependencies can execute in parallel. Second, the small number of abstractions that the programmer has to deal with are familiar from sequential programming. In particular, there are no new linguistic constructs for communication or synchronization. Finally, composing parallel programs by interconnecting sequential computations allows automatic support for heterogeneity and fault tolerance and facilitate software reuse as discussed in subsequent sections.

2.1 Computation Nodes

The basic computational unit of a Paralex program is a *multi-function* mapping some number of inputs to outputs. The graphical representation for the multi-function itself is a *node* and that of the inputs and outputs are incoming and outgoing links, respectively. The semantics associated with this graphical syntax obeys the so-called "strict enabling rule" of data-driven computations in the sense that when all of the incoming links contain values, the computation associated with the node starts execution transforming the input data to outputs. Paralex functions must be "pure" in that they can have no side effects. In particular, persistent internal state or interactions with external components such as files, devices and other computations are not permitted. Motivation and implications of this restriction are discussed in Section 8.

The actual specification of the computation may be done using whatever appropriate notation is available, including sequential programming languages such as C, C++, Fortran, Pascal, Modula or Lisp[1]. It is also possible for computations to be carried out through compiled binaries or library functions subject to architectural compatibility as discussed in Section 3.3.

How multiple output values are specified for multi-functions depends on the language being used to program the nodes. One possibility is to implement the functions as procedures and return results through call-by-reference parameters. Another possibility is to use simple functions and pack multiple results into composite data types such as structures, records or arrays. We pursue this option in the next section.

2.2 Filter Nodes

Filters permit multi-functions to be implemented using simple functions. They allow the single (structured) result to be picked apart to produce multiple outputs. In this manner, subsets of the data produced by the function may be sent to different destinations in the computation graph. This is a principal paradigm for data-parallel computing. For example, a single large matrix produced by some node in the computation may be "filtered" by extracting each of the quadrants to produce four sub-matrixes to be processed in parallel at the next level.

Conceptually, filters are defined and manipulated just as regular nodes and their "computations" are specified through sequential programs. In practice, however, all of the data filtering computations are executed in the context of the single process that produced the data rather than as separate processes. Associating filters with the producer of the data not only saves network bandwidth, it also economizes on data buffers necessary at the consumers.

Note that if node computations included explicit constructs for communication, outputs could be sent directly to their destinations as they were being produced and filters would be unnecessary. This, however, would be contrary to our goal of "plugging in" existing sequential code into the nodes without having to extend them with new language constructs. In the case of complied binaries or library functions where source code is not available, such extensions may not even be physically possible.

2.3 Subgraph Nodes

Paralex computation graphs may be structured hierarchically. Any node may contain a graph structure rather than sequential code to carry out its computation. These *subgraph* nodes obey the same semantics as primitive multi-function nodes and may be further decomposed themselves. Subgraphs are to Paralex what procedures are to sequential programs — a structuring abstraction that renders programs not only easier to understand, but also easier to construct using pre-programmed components.

[1] The exact list of programming languages permissible for node computations is determined by type and calling convention compatibility with C

2.4 Cycle Nodes

At the inter-node level, Paralex computation graphs are acyclic. Any single node of the graph, however, can be invoked repetitively during execution as long as its outputs exactly match its inputs (in number and type). The termination condition for the cycle can be dynamic as it is defined explicitly by the user as a function of the node inputs.

The Paralex cycle construct has a "while-loop" semantics and operates as follows. If the termination function evaluates to false, the node computation is skipped and the input values appear as output. Otherwise, the outputs produced by the node computation are "wrapped around" and become inputs for the next iteration. While the cycle is active, external input to the node is blocked.

3 Overview of Paralex

Paralex consists of four logical components: A graphics editor for program development, a compiler, an executor and a runtime support environment. The first three components are integrated within a single graphical programming environment. It is, however, possible to edit, compile or execute Paralex programs from machines with no graphics support. In this section we illustrate some of the principal characteristics of the user interface through examples.

3.1 Site Definition

The distributed system on which Paralex is to run is defined through a file called **paralex.site**. This file can be viewed as a crude "architecture description database." Each host of the system that is available for Paralex has a single-line descriptor. The first field of the line is the name of the host. Following the name is a comma-separated list of attributes. An example of a site file is shown in Figure 1.

```
elettra     sparc, SunOS, rel=4, rev=1, fpu, graphics=2, color, specmarks=21
leporello   sparc, SunOS, rel=4, rev=1, fpu, specmarks=13
dongiovanni sparc, SunOS, rel=4, rev=1, fpu, graphics, specmarks=10
nabucco     mips, Ultrix, rel=4, fpu, graphics, specmarks=18
tosca       m68020-SunOS, SunOS, rel=4, rev=0, fpu, specmarks=9
violetta    m68020-A/UX, specmarks=5
turandot    R6000, AIX, fpu, specmarks=12
carmen      vax, Ultrix, rel=4, graphics, memory=16, specmarks=6
jago        dongiovanni
```

Figure 1: Paralex Site Definition File.

The attributes may be binary (e.g., **sparc**, **fpu**) or numeric (e.g., **graphics**, **specmarks**). Including the name of a binary attribute for a host signals its presence. Numeric attributes are associated integer values through assignment and are set to one by default. Indicating another host name as an attribute permits the two hosts to share the same set of attributes (e.g., jago has the same attributes as dongiovanni). Paralex neither defines nor interprets keywords as attribute names. They are used by the Paralex loader to select sets of hosts suitable for executing nodes through a pattern matching scheme. This mechanism allows new attributes to be introduced and old ones to be modified at will by the user.

A minimum description of a host must contain its name, the processor architecture family (e.g., sparc, mips, vax) and the raw processor power measured in SPECmarks [22]. In case two hosts share the same processor architecture family but are not binary compatible, additional information (such as the operating system type, SunOS, A/UX) must be included in the characterization to distinguish them. The SPECmark value of a host is used by the mapping and dynamic load balancing algorithms to associate computations with hosts.

Note that the site definition contains no explicit information about the communication characteristics of the distributed system. The current version of Paralex assumes that each pair-wise communication between hosts is possible and uniform. This assumption is supported by broadcast LAN-based systems that are of immediate interest to us. With the advent of gigabit wide-area networking technologies [30], the site definition file could easily be extended to include explicit communication topology information and permit parallel computing over non-homogeneous and long-haul communication networks.

3.2 The Graphics Editor

The Paralex graphics editor allows computation graphs to be constructed in a simple and natural manner. The user interface is based on interactions between a pointing device such as a mouse and pull-down menus. An actual screen image of a Paralex session is displayed in Figure 2. The application being programmed is a parallel solution to the Synthetic Aperture Radar (SAR) problem where radar echo data collected from an aircraft or spacecraft flying at a constant altitude are used to reconstruct contours of the terrain below despite cloud cover, speed fluctuations, etc. The steps necessary for producing high-quality images from SAR data consist of the following sequence of computations: two-dimensional discrete Fourier transform, binary convolution, two-dimensional inverse discrete Fourier transform and intensity level normalization for visualization.

As can be seen from the Figure, the application consists of a top-level graph called SAR and a number of subgraphs corresponding to the two-dimensional discrete Fourier transform, convolution, two-dimensional inverse discrete Fourier transform and intensity level normalization computations. Nodes that are subgraphs or cycles can be *opened* to reveal their structure in separate windows. Primitive nodes, subgraphs and cycles can be distinguished by their graphical representation. Function outputs are designated as small bubbles attached to a node. They act as place holders for naming the output and associating a filter with it. Links originating from the same bubble carry the same data. In their expanded form, subgraphs and cycles have their inputs and outputs named through *anchors* drawn as gray (virtual) nodes. Finally, multiple links between a pair of nodes can be *bundled* together to eliminate visual clutter. This is the case for all of the links in the top-level graph SAR.

The graphics editor implements a small number of primitives for creating, deleting, repositioning nodes and links. The graphics editor also permits cut-and-paste operations and undoing the last

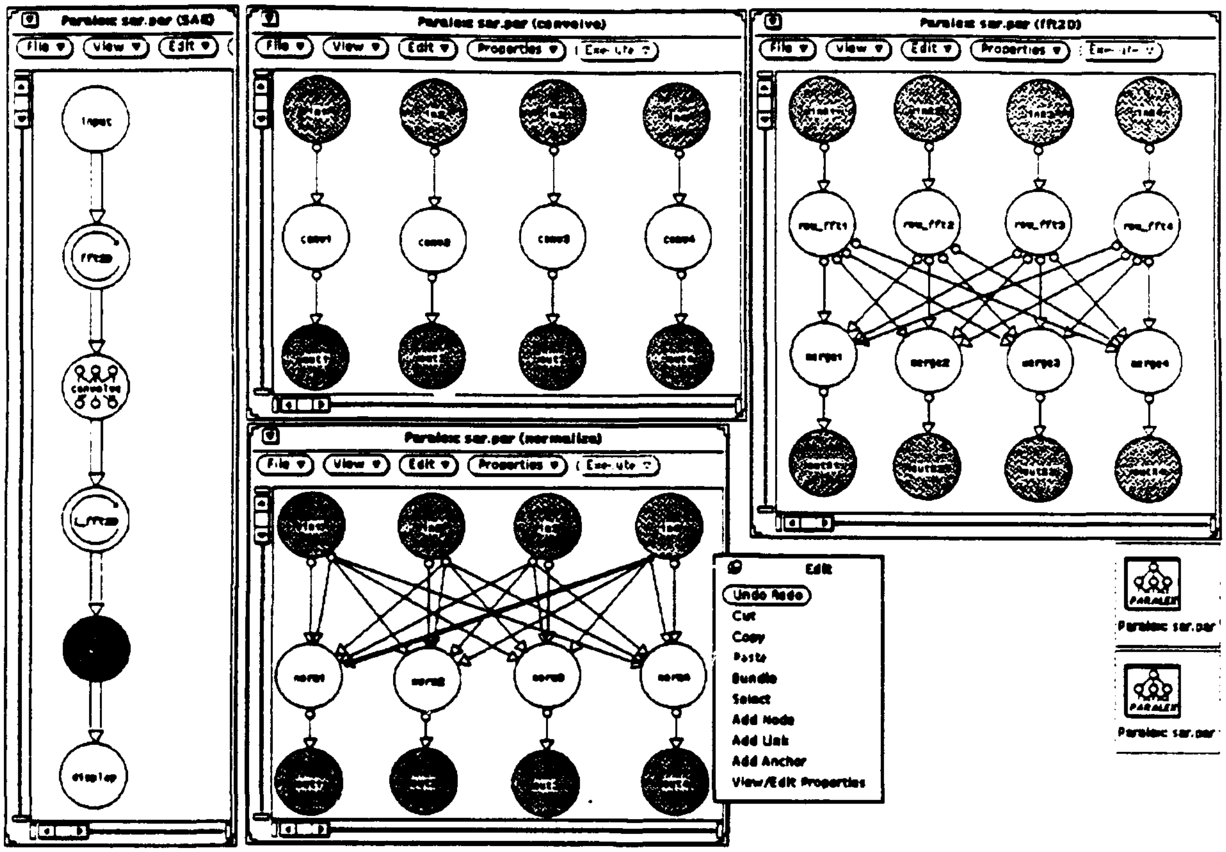


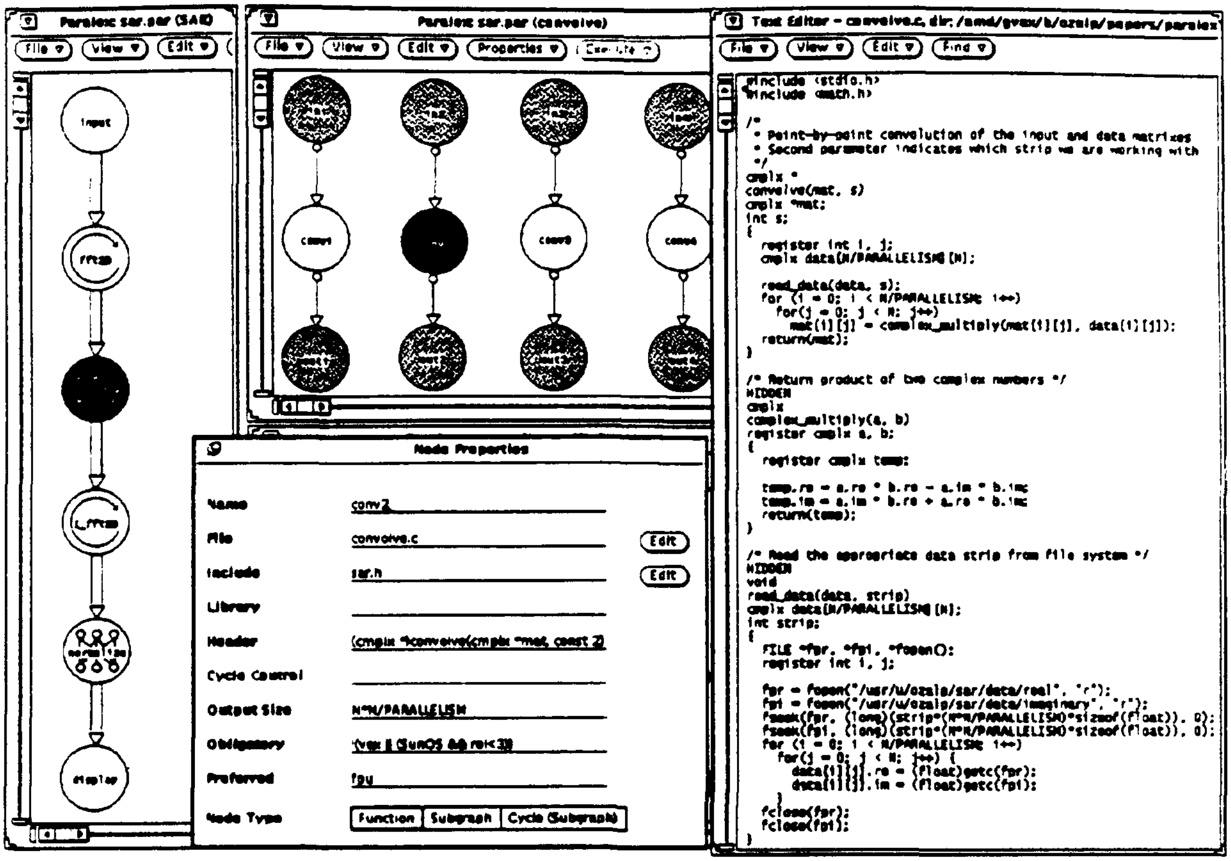
Figure 2: Paralex Graphics Editor.

edit function performed. The current computation can be saved for future use or an existing graph can be loaded for inclusion in the program. Paralex saves program graphs as plain text files with a C-like syntax for describing the attributes and structure. This permits Paralex programs to be manipulated, compiled and executed from machines that do not have graphics support.

3.3 Specifying Program Properties

Next stage in program development is to specify the properties of the various program components. This is accomplished by filling in property charts called *panels*. At the application level, the only property to specify is the fault tolerance degree. As described in Section 4, Paralex will use this information to automatically replicate the application. At the node level, the properties to be specified are more numerous.

Figure 3 displays the node panel associated with the highlighted node of subgraph conv where the following fields can be specified:



```
#include <stdio.h>
#include <math.h>

/*
 * Point-by-point convolution of the input and data matrixes
 * Second parameter indicates which strip we are working with
 */
cmplx *
convolve(mat, s)
cmplx *mat;
int s;
{
  register int i, j;
  cmplx data[N/PARALLELISM][N];

  read_data(data, s);
  for (i = 0; i < N/PARALLELISM; i++)
    for(j = 0; j < N; j++)
      mat[i][j] = complex_multiply(mat[i][j], data[i][j]);
  return(mat);
}

/* Return product of two complex numbers */
HIDDEN
cmplx
complex_multiply(a, b)
register cmplx a, b;
{
  register cmplx temp;

  temp.re = a.re * b.re - a.im * b.im;
  temp.im = a.im * b.re + a.re * b.im;
  return(temp);
}

/* Read the appropriate data strip from file system */
HIDDEN
void
read_data(data, strip)
cmplx data[N/PARALLELISM][N];
int strip;
{
  FILE *fpr, *fpi, *fopen();
  register int i, j;

  fpr = fopen("/usr/u/ozalp/sar/data/real", "r");
  fpi = fopen("/usr/u/ozalp/sar/data/imaginary", "r");
  fseek(fpr, (long)(strip*(N*N/PARALLELISM)*sizeof(float)), 0);
  fseek(fpi, (long)(strip*(N*N/PARALLELISM)*sizeof(float)), 0);
  for (i = 0; i < N/PARALLELISM; i++)
    for(j = 0; j < N; j++) {
      data[i][j].re = (float)getc(fpr);
      data[i][j].im = (float)getc(fpi);
    }
  fclose(fpr);
  fclose(fpi);
}
```

Name	conv2
File	convolve.c
Include	sar.h
Library	
Header	(cmplx *)convolve(cmplx *mat, const 2)
Cycle Control	
Output Size	N*N/PARALLELISM
Obligatory	(vax || (SunOS && rel<3))
Preferred	fpu
Node Type	Function | Subgraph | Cycle (Subgraph)

Figure 3: Specifying Node Properties and Computations.

Name Used to identify the node on the screen.

File Name of the file containing the computation associated with the node. It may be a source file, compiled binaries or a library.

Include Name of a file to be included before compilation. Used to include user-defined constants and data types in a Paralex program.

Header The interface specification for the function computed by the node. Must declare all input parameters and results along with their types. The syntax used is ANSI C with an extension to permit multiple return values for functions. Note that ANSI C is used only as an interface specification language and has no implications on the language in which the function is programmed.

Cycle Control Declaration of a Boolean function to be used for controlling cycle termination. The decision to continue the cycle can be a function of the same inputs that the computation is based on.

Output Size In case the dimension of an output structured type is not evident from the header declaration, it must be indicated here. For instance, this is the case for C arrays declared as pointers.

Obligatory A Boolean query that must be satisfied in order for the node to execute on a host. Used primarily for including compiled binaries or library functions of a specific architecture as node computations.

Preferred Further constrains the above list of hosts for performance reasons. The Paralex loader uses this information as a hint when making mapping decisions.

Node Type Identifies the node as one of `Function`, `Subgraph` or `Cycle`.

Obligatory and preferred host specifications are done using the site definition file described in Section 3.1. A query is formulated using C Boolean expression syntax. As an example, the obligatory query

```
!(vax || (SunOS && rel<3))
```

of Figure 3 prevents the node from executing on any host that either has the Vax architecture or is running a release of SunOS earlier than 3.0. Recall that the semantics associated with host attributes are user defined and Paralex uses them simply as tokens in constructing sets of hosts.

The obligatory host field should be used to indicate the weakest requirements for architectural and functionality reasons while the preferred host field typically adds performance-related queries. In case a particular query produces an empty set of candidate hosts for a particular site, the graphics editor signals a warning. A warning is also generated if the desired fault tolerance for the application cannot be satisfied with the given site definition. We consider these events warnings rather than errors since the binding of computations to hosts is not performed until load time. Given that Paralex programs may be transported and executed on systems other than those where they were developed, the warnings at edit time may be irrelevant at load time.

Clicking on the `Edit` button in the node panel invokes a text editor on the source file or the include file of a node. Figure 3 shows such an editor invoked on the file `convolve.c` containing the source code for the highlighted node "conv2". In this example, the function is programmed in C and makes use of two internal functions to perform the point-by-point convolution between its input and a data matrix read from the file system. Note that the code is ordinary sequential C and contains nothing having to do with remote communication or fault tolerance.

The attributes of a filter are specified and edited exactly in the same manner as an ordinary node through a panel associated with the output bubble. The only difference is that most of the fields in the filter panel are inherited from the parent node and are not modifiable. Finally, a link panel is used to name the input parameter of the destination function that is to receive the data value.

3.4 Compiling Paralex Programs

Once the user has fully specified the Paralex program by drawing the data flow graph and supplying the computations to be carried out by the nodes, the program can be compiled. The graphics editor saves the structural information of a Paralex program as plain text with a simple syntax. The textual representation of the Paralex program along with all of the source code for the node computations are fed as input to the compiler. Although the compiler may be invoked manually as a command, it is typically invoked from the graphical interface where the program was composed.

The first pass of the Paralex compiler is actually a precompiler to generate all of the necessary stubs to wrap around the node computations to achieve data representation independence, remote communication, replica management and dynamic load balancing. Type checking across links is also performed in this phase. Currently, Paralex generates all of the stub code as ordinary C. As the next step, a collection of standard compilers are invoked: C compiler for the stubs, perhaps others for the node computations. For each node, the two components are linked together to produce an executable module.

The compiler must also address the two aspects of heterogeneity — data representation and instruction sets. Paralex uses the ISIS toolkit [11, 12] as the infrastructure to realize a universal data representation. All data that is passed from one node to another during the computation are encapsulated as ISIS messages. Paralex automatically generates all necessary code for encoding-decoding basic data types (integer, real, character) and linearizing arrays of these basic types. The user must supply routines to linearize all other data types.

Heterogeneity with respect to instruction sets is handled by invoking remote compilations on the machines of interest and storing multiple executables for the nodes. Versions of the executable code corresponding to the various architectures are stored in subdirectories (named with the architecture class) of the current program. A network file server that is accessible by all of the hosts acts as the repository for the executables.

3.5 Executing Paralex Programs

The Paralex executor consists of a loader, controller and debugger. The debugger is incorporated into the graphical interface and uses the same display as the editor. It is described in [5]. The loader takes the output of the compiler and the textual representation of the computation graph as input and launches the program execution in the distributed system. As with the compiler, the loader can be invoked either manually as a command or through the graphical interface.

Before a Paralex program can be executed, each of the nodes (and their replicas, in case fault tolerance is required) must be associated with a host of the distributed system. Intuitively, the goals of this *mapping problem* are to improve performance by maximizing parallel execution and minimizing remote communication, to distribute the load evenly across the network, and to satisfy the fault tolerance and heterogeneity requirements. Since an optimal solution to this problem is computationally intractable, Paralex bases its mapping decisions on simple heuristics described in [4].

The units of our mapping decision are *chains* defined as sequences of nodes that have to be executed sequentially due to data dependence constraints. The initial mapping decisions, as well as modifications during execution, try to keep all nodes of a chain mapped to the same host. Since, by definition, nodes along a chain have to execute sequentially, this choice minimizes remote communication without sacrificing parallelism. Each node is executed as a Unix process that contains both the computation for the node and all of its associated filters.

4 Fault Tolerance

One of the primary characteristics that distinguishes a distributed system from a special-purpose super computer is the possibility of partial failures during computations. As noted earlier, these may be due to real hardware failures or, more probably, as a consequence of administrative interventions. To render distributed systems suitable for long-running parallel computations, automatic support for fault tolerance must be provided. The Paralex run-time system contains the primitives necessary to support fault tolerance and dynamic load balancing.

As part of the program definition, Paralex permits the user to specify a fault tolerance level for the computation graph. Paralex will generate all of the necessary code such that when a graph with fault tolerance k is executed, each of its nodes will be executed on $k + 1$ distinct hosts to guarantee success for the computation despite up to k failures. Failures that are tolerated are of the benign type for processors (i.e., all processes running on the processor simply halt) and communication components (i.e., messages may be lost). There is no attempt to guard against more malicious processor failures nor against failures of non-replicated components such as the network interconnect.

Paralex uses passive replication as the basic fault tolerance technique. Given the application domain (parallel scientific computing) and hardware platform (networks of workstations), Paralex favors efficient use of computational resources over short recovery times in its choice of a fault tolerance mechanism. Passive replication not only satisfies this objective, it provides a uniform mechanism for dynamic load balancing through late binding of computations to hosts as discussed in Section 5.

Paralex uses the ISIS *coordinator-cohort* toolkit to implement passive replication. Each node of the computation that requires fault tolerance is instantiated as a process group consisting of replicas for the node. One of the group members is called the *coordinator* in that it will actively compute. The remaining members are *cohorts* and remain inactive other than receiving broadcasts addressed to the group. When ISIS detects the failure of the coordinator, it automatically promotes one of the cohorts to the role of coordinator.

Data flow from one node of a Paralex program to another results in a broadcast from the coordinator at the source group to the destination process group. Only the coordinator of the destination node will compute with the data value while the cohorts simply buffer it in an input queue associated with the link. When the coordinator completes computing, it broadcasts the results to the process groups at next level and signals the cohorts (through another intra-group broadcast) so that they can discard the buffered data item corresponding to the input for the current invocation. Given that Paralex nodes implement pure functions and thus have no internal state, recovery from a failure is trivial — the cohort that is nominated the new coordinator simply starts computing with the data at the head of its input queues.

Figure 4 illustrates some of these issues by considering a 3-node computation graph shown at the top as an example. The lower part of the figure shows the process group representation of the

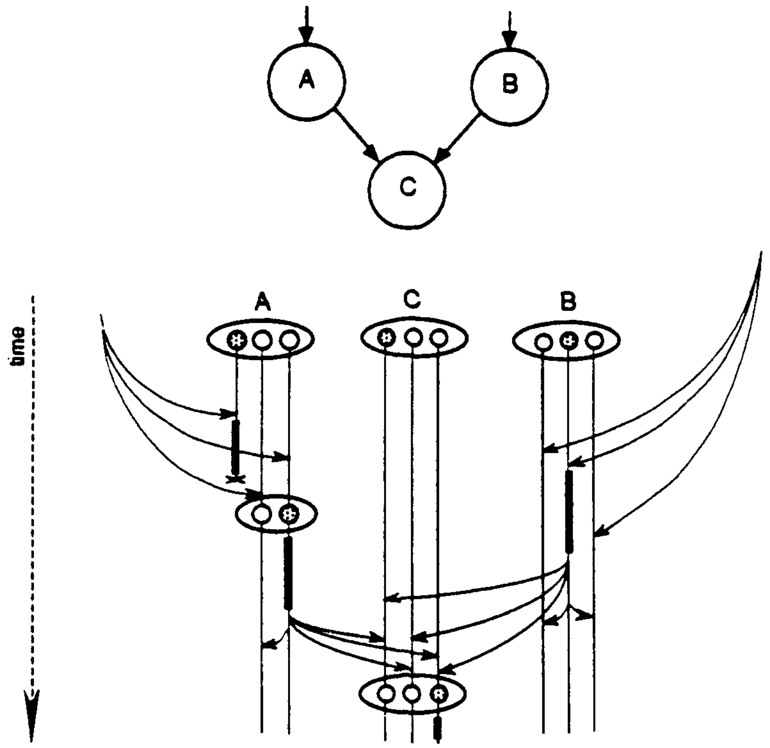


Figure 4: Replication and Group Communication for Fault Tolerance.

nodes based on a fault tolerance specification of 2. Arrows indicate message arrivals with time running down vertically. The gray process in each group denotes the current coordinator. Note that in the case of node A, the initial coordinator fails during its computation (indicated by the X). The process group is reformed and the right replica takes over as coordinator. At the end of its execution, the coordinator performs two broadcasts. The first serves to communicate the results of the computation to the process group implementing node C and the second is an internal group broadcast. The cohorts use the message of this internal broadcast to conclude that the current buffered input will not be needed since the coordinator successfully computed with it. Note that there is a small chance the coordinator will fail after broadcasting the results to the next node but before having informed the cohorts. The result of this scenario would be multiple executions of a node with the same (logical) input. This is easily prevented by tagging each message with an iteration number and ignoring any input messages with duplicate iteration numbers.

The execution depicted in Figure 4 may appear deceptively simple and orderly. In a distributed system, other executions with inopportune node failures, message losses and event orderings may be equally possible. What simplifies the Paralex run-time system immensely is structuring it on top of ISIS that guarantees "virtual synchrony" with respect to message delivery and other asynchronous events such as failures and group membership changes. Paralex cooperates with ISIS toward this goal by using a reliable broadcast communication primitive that respects causality [34].

5 Dynamic Load Balancing

To aɩhieve failure independence, each member of a process group representing a replicated node must be mapped to a different host of the distributed system. Thus, the computation associated with the node can be carried out by any host where there is a replica. To the extent that nodes are replicated for fault tolerance reasons, this mechanism also allows us to dynamically shift the load imposed by Paralex computations from one host to another.

As part of stub generation, Paralex produces the appropriate ISIS calls so as to establish a coordinator for each process group just before the computation proper commences. The default choice for the coordinator will be as determined by the mapping algorithms at load time. This choice, however, can be modified later on by the Paralex run-time system based on changes observed in the load distribution on the network. This situation is depicted in Figure 4 where the coordinator for the process group representing node C is switched from the left (default) to the right replica just before execution begins. By delaying the establishment of a coordinator to just before computation, we effectively achieve dynamic binding of nodes to hosts, to the extent permitted by having replicas around. Details of the run-time system in support of dynamic load balancing can be found in [5].

Perhaps the most dramatic effects of our dynamic load balancing scheme are observed when the computation graph is executed not just once, but repetitively on different input data. This so-called "pipelined operation" offers further performance gains by overlapping the execution of different iterations. Whereas before, the nodes of a chain executed strictly sequentially, now they may all be active simultaneously working on different instances of the input. In case of replicated chains, the dynamic load balancing mechanisms effectively spread the computational work among various hosts and achieve "vertical" parallelism.

6 Performance Results

The Synthetic Aperture Radar application described in Section 3.2 was compiled and run on a network of Sun-4/60 (SparcStation 1) hosts, each with 16 Megabytes of real memory. The communication medium was a standard 10-Megabit Ethernet connecting several hundred other hosts typical of an academic installation.

Experiments were performed to compare the performance of the 4- and 8-way parallel implementations of SAR with that of a sequential version coded in sequential C . The resulting speedup is displayed in Figure 5 as a function of the input dimension ranging from 64 × 64 to 1024 × 1024 matrixes. The experiments were conducted with the hosts configured and running as if in normal use but with no external users logged in. The hosts involved in the experiment were not isolated from the rest of the network, which continued to serve the normal Departmental traffic. The program was run with fault tolerance set to zero, thus disabling replication and prohibiting the possibility for dynamic load balancing. The mapping for the parallel executions were as follows: the input and output (display) nodes mapped to one host while the internal nodes, grouped into vertical chains, mapped to a different host associated with that chain. This resulted in the 4- and the 8-way parallel implementations using 5 and 9 hosts, respectively.

The results confirm our expectations — the larger the data size, the more significant the speedup.

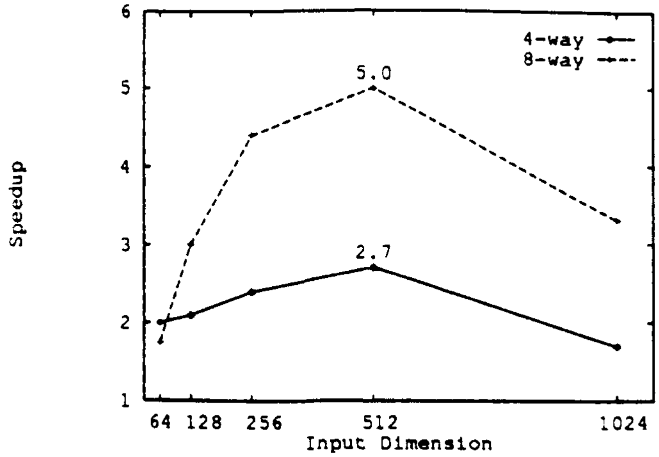


Figure 5: SAR Speedup for 4-way (5 Processors) and 8-way (9 Processors) Parallelism.

The fact that performance degrades for the largest input size is totally an artifact of thrashing due to paging. Note that in the case of a 1024 × 1024 image, the input data (radar signal and filter matrixes of complex numbers with 8 bytes per entry) amount to more than 16 Megabytes. Furthermore, even in the parallel versions, the input and output nodes are strictly sequential and have to process the entire radar signal and gray-scale matrixes, respectively. As a consequence, the host to which they are mapped has to support a demand for virtual memory significantly greater than the available 16 Megabytes of real memory, resulting in severe paging. In the case of small input sizes, it hardly pays to parallelize. The overhead of communication and process management outweigh the gains due to overlapped computing, which are minimal for these dimensions.

7 Related Work

Many of the goals of Paralex are shared by other systems that have been built or are currently under development. What is unique about Paralex, however, is the automatic support for fault tolerance and dynamic load balancing it provides and the extent to which the components of the system have been integrated into a single programming environment. For each Paralex design objective, we discuss other projects that share it or that have inspired our particular approach in dealing with it.

7.1 Network of Workstations as Multiprocessors

The idea of viewing a collection of workstations on a network as a parallel multiprocessor appears to be a popular one. There are numerous projects that have been experimenting with different abstractions to provide on top of such a system.

The Amber System [19] creates a shared memory multiprocessor abstraction over a network of Firefly multiprocessor workstations. The system is programmed using the shared objects paradigm with Amber handling all remote invocation and consistency issues.

The idea of broadcast-based parallel programming is explored in the **nigen++** system [2]. Using ISIS as the broadcast mechanism, **nigen++** supports a programming style not unlike the Connection Machine — a single master process distributes work to a collection of slave processes. This paradigm has been argued by Steele to simplify reasoning about asynchronous parallel computations without reducing their flexibility [39].

The master/slave model forms the basis of the AERO system [3]. Unlike **nigen++**, however, the model is completely asynchronous with the master having to explicitly accept results from parallel slave computations.

The Mentat system [27] is based on an object-oriented programming language derived from C++. The user encapsulates data and computation as objects and the compiler performs all the necessary data flow analysis to permit parallel invocations whenever possible. The run-time system ensures correct object semantics even though all invocations are performed asynchronously.

Yet another object-oriented approach to parallel programming is TOPSYS [9]. The basic programming abstraction in TOPSYS consists of tasks, mailboxes and semaphores realized through library routines. The emphasis of the system is a collection of graphical tools for performance monitoring and debugging.

The system that perhaps comes closest to Paralex in its design goals and implementation is HeNCE [8]. In HeNCE, the graphical representation of a computation captures the precedence relations among the various procedures. Data flow is implicit through syntactic matching of output names to parameter names. HeNCE graphs are dynamic in that subgraphs could be conditionally expanded, repeated or pipelined. Unlike Paralex, HeNCE has no automatic support for fault tolerance or dynamic load balancing.

7.2 Language and Architecture Heterogeneity

Our use of stubs for remote communication and universal data representation as ISIS messages derive their inspiration from Remote Procedure Call (RPC) systems [37, 13]. Our use of these techniques, however, is to permit flow of data across (potentially) heterogeneous architectures rather than flow of control. The semantics of the remote invocation in Paralex might be called "remote function call without return" in the sense that a node supplies (partial) input to another node only upon completion. Results of the remote computation are passed on to other nodes rather than being returned to the invoker. Some notable systems where RPC has been employed to permit mixed-language programming and support for heterogeneous architectures include Horus [26], Mercury [36], MLP [28] and HRPC [10].

7.3 Graphical Programming

Examples of systems that use graphical notations to express parallel computations include Fel [31], Poker [38], CODE [15], Alex [33], LGDF [7] and apE [23]. None of these systems addresses fault tolerance nor provides a programming environment in the sense of Paralex. Of these, perhaps CODE comes closest to Paralex in design goals. In addition to a graphical notation for parallel programs, it supports software reuse through a subsystem called ROPE [16]. The programming paradigm of CODE is based on the model of Browne [14] where programs are represented as generalized dependency graphs with each node having three sets of dependencies: input data. output data and exclusion. LGDF proposes a graphical notation based on large-grain data flow is very similar to that of Paralex but lacks a programming environment and run-time support system. apE is an interesting system that shares with Paralex the data flow model of computation but is limited to scientific data visualization applications.

7.4 Architecture Independence

As with any other language proposal, Paralex strives for architecture independence. This is particular true in the case of parallel programming languages since the competing models for parallel computation are still far too numerous. While the von Neumann architecture serves as the universal target for sequential programming languages, there exists no counterpart for parallel programming languages. There are two ways to address this problem: Propose a model of parallel computation and hope that it will be accepted as "the" model, or propose programming languages that can be efficiently mapped to a variety of competing models. Recent proposals by Valiant [40] and Steele [39] fall into the first category. Systems and notations such as Paralex, Par [20], UNITY [18], Linda [25], CODE, P^3L [21], Prelude [41] and Phase Abstractions [35] fall into the second camp. In the case of Paralex, we inherit the properties of the data flow notation and keep further goals for architecture independence rather modest. Within the class of MIMD architectures, we strive for independence from synchrony attributes and communication mechanisms.

8 Discussion

Paralex does not require a parallel or distributed programming language to specify the node computations. The graphical structure superimposed on the sequential computation of the nodes effectively defines a "distributed programming language." In this language, the only form of remote communication is through passage of parameters for function invocation and the only form of synchronization is through function termination. While the resulting language may be rather restrictive, it is consistent with our objectives of being able to program distributed applications using only sequential programming constructs. This in turn facilitates reusability of existing sequential programs as building blocks for distributed parallel programs. We also note that this programming paradigm is in the spirit of that proposed by Steele [39] where a severely-restricted programming paradigm is advocated even for environments that support arbitrary asynchrony.

Paralex computation graphs are static. The only source of dynamism in the computation is the cyclic execution of a subgraph. This same mechanism could also be used to realize conditional subgraph expansion. Other dynamic structures such as variable-degree horizontal expansion or pipelined subgraphs are not possible. While this certainly restricts the expressiveness of Paralex as a language, we feel that it represents a reasonable compromise between simplicity expressive power. Increasing the dynamism beyond that present would require Paralex to become a full-fledged functional programming language with all of the associated complexities.

While rendering failure recovery trivial, the requirement that Paralex functions be pure with no side effects makes certain programming tasks cumbersome. For example, cyclic nodes that base their computation on iteration number have to implement such a counter explicitly by passing it as a parameter. Furthermore, the lack of persistent internal state makes nodes that iterate over large data structures inefficient since the entire structure must be passed from one iteration to the next. Persistent internal state could be easily added to Paralex nodes. The internal group broadcast by the coordinator to the cohorts, currently used to signal successful completion, will have to be augmented to communicate the updated values for the persistent variables. Otherwise, the fault tolerance mechanisms remain unchanged. The decision to extend Paralex in this manner or not has is a philosophical one and pits functional purity against efficiency.

9 Status and Conclusions

A prototype of Paralex is running on a network of m680x0, SPARC, MIPS and Vax-architecture Unix workstations. Paralex relies on Unix for supporting ISIS and the graphical interface. The graphical interface is based on X-Windows with the Open Look Intrinsics Toolkit. As of this writing, the graphics editor, compiler and executor are functional. Dynamic load balancing and debugging support have been implemented but have not yet been integrated into the environment.

The current implementation has a number of known shortcomings. As computations achieve realistic realistic dimensions, executing each Paralex node as a separate Unix process incurs a large amount of system overhead. In the absence of shared libraries, each process must include its own copy of the ISIS services. This, plus the memory required to buffer each input and output of a node contribute to large memory consumption. We are working on restructuring the system to create a single Paralex process at each host and associate separate threads (ISIS tasks) with each node of the computation. In this manner, the library costs can be amortized over all the nodes and buffer memory consumption can be reduced through shared memory. Yet another limitation of the current implementation is its reliance on NFS as the repository for all executables. Obviously, the NFS server becomes a bottleneck not only for fault tolerance, but also for performance since it must supply the initial binary data and then act as a paging and swapping server for a large number of hosts. A satisfactory solution to this problem requires basing the storage system on a replicated file service such as Coda [32] or Echo [29].

Paralex provides evidence that complex parallel applications can be developed and executed on distributed systems without having to program at unreasonably low levels of abstraction. Many of the complexities of distributed systems such as communication, synchronization, remote execution, heterogeneity and failures can be made transparent automatically. Preliminary results indicate that the costs associated with this level of transparency are completely acceptable. We were able to obtain performances for the SAR application that achieved maximum speedups of 2.7 and 5.0 for the 4-way and 8-way parallel versions, respectively. Given the large amount of sequential code for reading in the problem input and displaying the result, these figures are probably not too far from the theoretical maximum speedup possible. We should also note that SAR represents a "worst-case" application for Paralex since its computation to communication ratio is very low. Most importantly, however, the performance results have to be kept in perspective with the ease with which the application was developed.

Paralex is an initial attempt at tapping the enormous parallel computing resource that a network of workstations represents. Further experience is necessary to demonstrate its effectiveness as a tool to solve real problems.

Acknowledgements Giuseppe Serazzi and his group at the University of Milan contributed to early discussions on the mapping and dynamic load balancing strategies. Ken Birman and Keshav Pingali of Cornell University were helpful in clarifying many design and implementation issues. Dave Forslund of Los Alamos provided valuable feedback on an early prototype of the system. Alberto Baronio, Marco Grossi, Susanna Lambertini, Manuela Prati and Nicola Samoggia of the Paralex group at Bologna contributed to the various phases of the coding. We are grateful to all of them.

References

[1] W. B. Ackerman. Data Flow Languages. *IEEE Computer,* February 1982, pp. 15–22.

[2] R. Anand, D. Lea and D. W. Forslund. Using **nigen++**. Technical Report, School of Computer and Information Science, Syracuse University, January 1991.

[3] D. P. Anderson. The AERO Programmer's Manual. Technical Report, CS Division, EECS Department, University of California, Berkeley, October 1990.

[4] Ö. Babaoğlu, L. Alvisi, A. Amoroso and R. Davoli. Mapping Parallel Computations onto Distributed Systems in Paralex. In *Proc. IEEE CompEuro '91 Conference*, Bologna, Italy, May 1991.

[5] Ö. Babaoğlu, L. Alvisi, A. Amoroso, R. Davoli and L. A. Giachini. Run-time Support for Dynamic Load Balancing and Debugging in Paralex. Technical Report, Department of Computer Science, Cornell University, Ithaca, New York, December 1991.

[6] R. G. Babb II. Parallel Processing with Large-Grain Data Flow Techniques. *IEEE Computer*. July 1984, pp. 55–61.

[7] R. G. Babb II, L. Storc and R. Hiromoto. Developing a Parallel Monte Carlo Transport Algorithm Using Large-Grain Data Flow. *Paralllel Computing,* vol. 7, no. 2, June 1988, pp. 187–198.

[8] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek and V. S. Sunderam. Graphical Development Tools for Network-Based Concurrent Supercomputing. In *Proc. Supercomputing '91*, November 1991, Albuquerque, New Mexico.

[9] T. Bemmerl, A. Bode, *et al.* TOPSYS — Tools for Parallel Systems. SFB-Bericht 342/9/90A, Technische Universität München, Munich, Germany, January 1990.

[10] B. N. Bershad, D. T. Ching, E. D. Lazowska, J. Sanislo and M. Schwartz. A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems. *IEEE Trans. on Software Engineering,* vol. SE-13, no. D82, August 1987, pp. 880–894.

[11] K. Birman, R. Cooper, T. Jcseph, K. Kane and F. Schmuck. The ISIS System Manual.

[12] K. Birman and K. Marzullo. ISIS and the META Project. *Sun Technology,* vol. 2, no. 3 (Summer 1989), pp. 90–104.

[13] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Trans. on Computer Systems*, vol. 2, no. 1, February 1984, pp. 39–59.

[14] J. C. Browne. Formulation and Programming of Parallel Computations: A Unified Approach. In *Proc. Int. Conf. Parallel Processing*, Los Alamitos, California, 1985, pp. 624–631.

[15] J. C. Browne, M. Azam and S. Sobek. CODE: A Unified Approach to Parallel Programming. *IEEE Software*, July 1989, pp. 10–18.

[16] J. C. Browne, T. Lee and J. Werth. Experimental Evaluation of a Reusability-Oriented Parallel Programming Environment. *IEEE Trans. on Software Engineering*, vol. 16, no. 2, February 1990, pp. 111–120.

[17] N. Carriero and D. Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, vol. 21, no. 3, September 1989, pp. 323–358.

[18] K. M. Chandy. Programming Parallel Computers. Technical Report Caltech-CS-TR-88-16. Department of Computer Science, California Institute of Technology, August 1988.

[19] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy and R. J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. University of Washington, Department of Computer Science Technical Report 89-04-01, April 1989.

[20] M. H. Coffin and G. R. Andrews. Towards Architecture-Independent Parallel Programming. Technical Report TR 89-21a, Department of Computer Science, University of Arizona, Tucson, Arizona, December 1989.

[21] M. Danelutto, R. Di Meglio, S. Pelegatti and M. Vanneschi. High Level Language Constructs for Massively Parallel Computing. In *Proc. Sixth International Symposium on Computer and Information Sciences*, Elsevier North-Holland, October 1991.

[22] K. Dixit. SPECulations: Defining the SPEC Benchmark. *SunTech Journal*, vol. 4, no. 1, January 1991, pp. 53–65.

[23] D. S. Dyer. A Dataflow Toolkit for Visualization. *IEEE Computer Graphics and Applications*, July 1990, pp. 60–69.

[24] E. Fairfield. Private communication. Los Alamos National Laboratory, New Mexico.

[25] D. Gelernter, N. Carriero, S. Chandran and S. Chang. Parallel Programming in Linda. In *Proc. Int. Conf. Parallel Processing*, St. Charles, Illinois, August 1985, pp. 255–263.

[26] P. B. Gibbons. A Stub Generator for Multilanguage RPC in Heterogeneous Environments. *IEEE Trans. Software Engineering*, vol. SE-13, no. 1, January 1987, pp. 77–87.

[27] A. S. Grimshaw. An Introduction to Parallel Object-Oriented Programming with Mentat. Technical Report No. TR-91-07, Department of Computer Science, University of Virginia, April 1991.

[28] R. Hayes, S. W. Manweiler and R. D. Schlichting. A Simple System for Constructing Distributed, Mixed-Language Programs. Technical Report TR 87-1b, Department of Computer Science, University of Arizona, Tucson, Arizona, February 1988.

[29] A. Hisgen, A. Birrell, C. Jerian, T. Man, M. Schroeder and G. Swart. Granularity and Semantic Level of Replication in the Echo File System, In *Proc. Workshop on Management of Replicated Data*, Houston, IEEE CS Press 2085, November 1990, pp. 5–10.

[30] IEEE Special Report. Gigabit Network Testbeds. *IEEE Computer*, vol. 23, no. 9, September 1990, pp. 77–80.

[31] R. M. Keller and W-C. J. Yen. A Graphical Approach to Software Development Using Function Graphs. In *Proc. Compcon Spring 1981*, CS Press, Los Alamitos, California, pp. 156–161.

[32] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File system. In *Proc. 13th ACM Symposium on Operating Systems Principles*, Asilomar, Pacific Grove, California, October 1991.

[33] D. Kozen, T. Teitelbaum, W. Chen, J. Field, W. Pugh and B. Vander Zanden. ALEX: An Alexical Programming Language. In *Visual Programming Languages*, Ed. Korfnage, Plenum Press.

[34] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, vol. 21, no. 7, July 1978, pp. 558–565.

[35] C. Lin and L. Snyder. Portable Parallel Programming: Cross Machine Comparisons for SIMPLE. In *Fifth SIAM Conference on Parallel Processing*, 1991.

[36] B. Liskov, *et al.* Communication in the Mercury System. In *Proc. Twenty-first Annual Hawaii Conference on System Sciences*, January 1988.

[37] S. K. Shrivastava and F. Panzieri. The Design of a Reliable Remote Procedure Call Mechanism. *IEEE Trans. Computers*, vol. C-31, no. 7, pp. 692–697.

[38] L. Snyder. Parallel Programming and the Poker Programming Environment. *IEEE Computer*, July 1984, pp. 27–36.

[39] G. L. Steele Jr. Making Asynchronous Parallelism Safe for the World. In. *Proc. 17th Annual ACM Symposium on Principles of Programming Languages*, 1990, pp. 218–231.

[40] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, vol. 33, no. 8, August 1990, pp. 103–111.

[41] W. Weihl, E. Brewer, A. Colbrook, C. Dellarocas, W. Hsieh, A. Joseph, C. Waldspurger and P. Wang. Prelude: A System for Portable Parallel Software. MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-519, October 1991.